JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 183–199 (1999)

Practice

Isolating Faults in Complex COTS-based Systems

SCOTT A. HISSAM* and DAVID CARNEY
Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213–3890, U.S.A.

SUMMARY

This paper provides an overview of isolating and overcoming faults in COTS-based systems. It describes a method and mechanisms that are useful for maintainers and integrators that are tasked with maintaining complex systems from heterogeneous sources. While all readers may find value in this report, it is expressly aimed at a technical audience. The method described in this paper has been used on various systems. Three uses of the method are described in the paper. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: COTS; component-based application; maintenance methods; program understanding; debugging

1. SOME REALITIES OF COTS-BASED SYSTEMS

There are many advantages claimed for using commercial off-the-shelf (COTS) components in modern systems. The first and most obvious advantage is lower total cost of ownership, but others are easily noted: immediacy of acquisition, greater ease of modernization and reduced development costs.

Despite these advantages, sometimes things 'go wrong' and faults (a.k.a. 'bugs') arise. This paper describes one method for debugging COTS-based systems, since methods used for traditional systems tend to be less useful (Beizer, 1995; Myers, 1977). The method is based on several real instances where systems were fielded, failed and needed correction. This paper describes the conceptual basis of the method, and suggests various mechanisms through which it can be used to isolate faults in complex, COTS-based systems.

2. WHY THINGS GO WRONG

2.1. Lack of visibility

There is an unavoidable truth that accompanies the decision to use COTS components: individual component evaluation will typically not illuminate all of a given component's shortcomings, no

*Correspondence to: Scott A. Hissam, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213–3890, U.S.A. Email: shissam@sei.cmu.edu

CCC 1040–550X/99/030183–17$17.50

Received 10 November 1998
Revised 19 March 1999

matter how extensively an assessment might be done. Further, as the complexity of a system *increases* (i.e., system capability, heterogeneity, number of components to be integrated, etc.), the likelihood that all components and their interactions can be accurately appraised *decreases* (Fox, Lantner and Marcom, 1997). This means that use of COTS components magnifies the problem that plagues all complex systems, namely, that system failure may require extensive and difficult debugging to isolate its source.

These failures can occur for many different reasons. When commercially-based systems fail, however, some circumstances (e.g., dependence on vendor priorities, loss of support for old releases) particular to the use of COTS components become apparent. Possibly the worst circumstance, as Voas (1998) points out, is the absence of source code, since many testing techniques rely on code availability (Chan and Lakhotia, 1998). Debugging becomes a different, and much more difficult, activity when source code is unavailable. Even the basic meaning of 'debugging' shifts. Debugging no longer centres on finding erroneous code (i.e., 'bugs'), but focuses on observing the behaviour of large, opaque components and making inferences from that behaviour.

Regardless of the actual cause of failure, the person who does this 'debugging' is not the person who created the component and knows its internals, but rather is the integrator who combined it with other components into the complex system, and this will be the focus of the present paper. The outcome of the integrator's 'debugging' is not correcting source code, but instead is determining whether, and how, the component's unexpected behaviour can be reconciled with the desired behaviour of the whole system.

Finally, even when the source of failure is isolated in a commercial component, getting the vendor to take notice and repair the fault may prove difficult or impossible. Many system integrators have found themselves at an impasse because a vendor is unable or unwilling to divert resources, change release schedules, or accommodate requests for corrections or improvements to the vendor's component.

These three points—insufficiency of evaluation techniques, system opaqueness and the vendor's response to trouble reports—can make the task of isolating and correcting faults in a COTS-based system more difficult. We examine each in turn.

2.2. Insufficiency of COTS software evaluation techniques

Current techniques for software evaluation are immature and unreliable. This is especially true of techniques for evaluating COTS products. Testing is not easily done, for instance, and a good amount of understanding of the component depends on the vendor's claims for the product. Another problem, as discussed by Carney and Wallnau (1998), in evaluating COTS components is that it is rare for two components to be fully comparable. More often the evaluator will be comparing two components, each of which satisfies only some subset of the desirable features, and either component may leave some requirements unsatisfied. The evaluator must juggle between conflicting imperatives, since the system's eventual customers seldom spell out the difference between 'hard' and 'soft' requirements.

A final problem in evaluating COTS components appears, paradoxically, only after the fact. By whatever process of elimination, some choice will be made between competing candidates. However, *after* a selection has been made, evaluation will likely cease. This means that any unidentified shortcomings in the chosen component will probably not be found until much later in

the system's life cycle. This may be at a point where previous purchasing or licensing commitments, or schedule factors, mandate the use of the component, no matter how severe the newfound shortcoming proves to be.

2.3. System opaqueness

A major complication for any system is the extent of its diversity. As the number of components to be integrated with each other grows, and as the number of combinations increases, then so does the likelihood that failures will occur in one of three possible places:

- in any individual component,
- in any interaction between pairs of components, and
- in the entire system itself.

If the system fails, fault isolation becomes progressively more difficult: is the failure caused by one component? If so, which? Is failure caused by an interaction of components? If so, which combination?

While these questions are applicable to any kind of system, the presence of commercial components adds a new dimension of complexity. To the integrator of a COTS-based system, isolating the fault entails first ascertaining *how* the components work and then finding out *why* they fail. This is the natural consequence of the integrator's lack of visibility into the components and lack of control over their workings.

The integrator may also lack insight into the design assumptions that the component's developers had. The integrator's view of the requirement being addressed, and the role the component has in implementing a solution, may be different from that of the component's developer. As an example: many POSIX-compliant operating systems support loadable device drivers. The initial intent of these device drivers was to allow new devices to be added without the necessity of rebuilding the operating system kernel. Most vendors assume that these device drivers will be loaded once upon booting the system. However, it is possible for a third-party integrator to base the design of his system on the ability to load and unload these drivers many times during system execution. This leads to a potential mismatch of assumptions: the vendor is not concerned with releasing resources (since the drivers were not expected to be reloaded without rebooting the system), and the integrator expects that drivers can be loaded frequently. This mismatch creates a situation where the device drivers may fail to reload, may fail to operate as expected or may corrupt system integrity.

2.4. A vendor's response to trouble reports

When a COTS-based system fails, the initial tendency is to rely on the COTS vendors to provide a solution. In most cases, the system integrator has the task of proving to one vendor that the at-fault component is the one provided by that vendor. When dealing with multiple vendors, it is very likely that each vendor will suggest that the fault lies with some other vendor's component and not with their own component. This is perhaps an understandable reaction: customer complaints are often unfounded, and a vendor's perceived inaction can be due to a lack of sufficient evidence upon which to act or other factors unseen by the integrator (Hybertson, Ta and Thomas, 1997).

However, when serious technical shortcomings in a particular COTS component exist, getting the component's vendor to correct the shortcomings is often the best way to achieve a solution. To bring this about, the integrator must collect and analyse as much hard evidence as possible. A typical first action is to call the vendor's support line (e.g., an 800 number). Presuming that the integrator gets to speak with a knowledgeable person (or still more unlikely, the actual developer), the integrator will need to describe the problem in as much detail as possible. At best, the integrator will eventually convince the component's developer that a correction is necessary, which unfortunately will typically not be introduced until some future release of the component. At worst, the component developer will refuse to make any change to the component. In either case, for the integrator, this means that a workaround must be found or the system cannot function.

In brief, if the system's failure is due to the interaction of two or more components, then it is very improbable that any of the vendors will be able to offer any substantive assistance. If the failure is due to a single component and the cause of failure can be clearly identified to the vendor, then it is possible that the vendor may provide a solution. But in either of these cases, it is far more likely that the integrator who put the system together will be the one who must eventually debug the system. And this requires a very different notion of 'debugging' than exists for a system whose source code is available.

3. A METHOD FOR DEBUGGING COTS-BASED SYSTEMS

The method we propose is not novel and we claim no rights of invention. Instead, after considerable introspection about the problems described above, we look backward at the way science has proceeded for centuries. We argue for a systematic approach for diagnosing and correcting failures in COTS-based systems that is essentially that of the classic scientific method—observation, hypothesis and prediction (Toulmin, 1987). An hypothesis is formed based on one or more observations. To test that hypothesis, a prediction is made relative to some stimulus and anticipated effect. An experiment is then designed to test the prediction. Results from the experiment will either support or contradict the hypothesis. If the hypothesis is not supported, refinement of the hypothesis and subsequent experimentation are needed. We illustrate this in Figure 1.

This method, having proved applicable in many sciences for many centuries, has equal validity today in software engineering, where traditional software debugging implicitly makes use of it. Suppose, for instance, that in some hypothetical software system, a problem report (i.e., the observation) indicates that a sorting routine fails when the number of elements in the list gets too big. An initial assumption (the hypothesis) is that the list is bounded by a hard-coded fixed limit; the prediction is that a review of the source code will reveal a constant that provides the list's upper boundary. If the ensuing review locates no such constant (thus contradicting the hypothesis) the hypothesis must be refined, and other debugging activities will begin.

This iterative sequence is very different when the system makes use of COTS components. Now the components are opaque: their *behaviour* can be observed but their inner workings are not visible, so predictions about their code have no meaning. To illustrate this point, assume the previous hypothetical scenario of a defective sorting routine. The first recourse described above is no longer available. Instead, it becomes necessary to isolate the problem by observing the system's behaviour in one or more different contexts. Thus, a hypothesis that conforms to this context (i.e., the failure of the sort function) might be that scarce memory resources adversely affect the sort

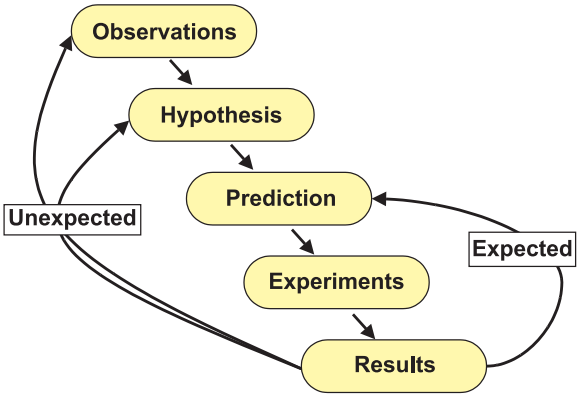


Figure 1. The scientific method

routine. A prediction could be made that by starving the system of available memory resources, the sort component will fail even when the number of elements in the list remain below that stated in the problem report. An experiment can then be built to consume available memory resources and re-run the sort component with a controlled data set; if the component fails as expected, then the hypothesis (i.e., that the sort component is affected by low memory conditions) has been verified.

Notice the different approaches used by these examples. When source code was available, it was a fairly straightforward process to determine whether a constant upper bound was the cause. If this was not the cause of the failure, then successive iterations could have included debug statements, code profilers, static and dynamic analysers, and other techniques readily available to maintainers. In the second case, however, such means are not available to the integrator. Now, the integrator must choose a diagnostic path that manipulates the environment in which the component operates, in order to observe cause and effect.

Given the assumption of this method (i.e., that there is something to observe) and the fact that with COTS components such observations—and the insights they provide—cannot rely on examining the source, it is obvious that other sources of visibility must be found. We now describe some mechanisms that provide the needed visibility into COTS-based systems.

4. MECHANISMS FOR COMPONENT AND SYSTEM OBSERVATION

4.1. Three visibility situations

There are many existing tools and techniques that are useful for providing visibility into COTS components. The choice of tool or technique is based on the kind of component and the type of suspected failure. If the component itself is suspect, the most useful mechanisms are those that can be used to ‘look inside’ the black box. If the failure is thought to be between two components, then the mechanisms will look at the interplay between the suspected components. Finally, if the entire system (or subsystem) has failed, then the techniques needed will be those that can assess a group

Table 1. Probing tools

Probing tool	System availability	Function
`ps`	UNIX and UNIX derivatives	Parent/child relationships, aggregate resource utilization
`trace/truss`	UNIX and UNIX derivatives	System calls, parameters, return codes, error codes
`crash`	UNIX and UNIX derivatives	Detailed resource utilization, signal disposition, open files, user stack
`Process Viewer`	WinNT/Win95/98	Memory allocation, processor utilization, thread information, API calls
`Spy/Spy++`	WinNT/Win95/98	View/manipulate message streams for a window, thread, or process

of components working in unison.

All of these mechanisms represent 'observation posts' that provide insight into off-the-shelf components:

- *Intra-component visibility*: tools that observe the behaviour of an individual component; these serve to highlight the component developer's assumptions about the intended use of the component.
- *Inter-component visibility*: tools that observe the behaviour of two or more components; these serve to highlight potential mismatches between components.
- *Extra-component visibility*: techniques to observe a system of components in ensemble with the desire to understand macro-level issues dealing with performance, maintenance, misfits, etc.

4.2. Intra-component visibility

Tools in this category are used to gain insight into the behaviour of an individual component *in itself*. This is generally accomplished by *probing*. Probing tools obtain visibility into parent/child relationships, thread performance, user stack, resource utilization, signal and event disposition, system calls (including parameters and return codes), and open files. Table 1 provides a sampling of tools available to perform some of these types of probes.

These tools let the observer see how a component makes use of native operating system resources. Since this information tends to be highly specific to individual operating systems, tools such as these are typically provided with the operating system or development environment. Reference material about these tools is generally available to help the integrator understand the tools' resources, values, limits, settings and codes. This material is commonly in the form of manual pages, documentation, API references, library references and help files.

These probing materials and tools help the integrator to get past the 'black box' status of off-the-shelf components, where all internal workings and internal behaviours of COTS products are hidden. Using these materials and tools can improve the integrator's understanding of a component's internal

Table 2. Snooping tools

Snooping tool	System availability	Function
`Etherfind` (a.k.a. `tcpdump`, a.k.a. `snoop`)	UNIX and UNIX derivatives	Capture, view and filter TCP/IP-based network traffic, view TCP and IP headers and data
`Ipcs`	UNIX and UNIX derivatives	Status and information about shared memory, message queues and semaphore use
`protocol analyser`	(Not OS specific)	Capture, view, and filter analyser-specific logical and physical network layers
`Object Viewer`	WinNT/Win95/98	Monitor OLE 2.0-enabled applications' interaction with the operating system
`DDESpy`	WinNT/Win95/98	Monitor dynamic data exchange in the operating system

behaviour and reveal inherent ambiguities, thereby increasing the ease with which a component can be integrated.

4.3. Inter-component visibility

Tools in the inter-component visibility category provide insight into the interactions between components. Such insight is generally accomplished by *snooping*. Snooping techniques are very often the same as those used for illegal and intrusive attacks on computer systems. Not surprisingly, these techniques yield insight and benefit to the integrator. Snooping tools obtain visibility into logical and physical protocol streams, state information, procedure calls and data exchange. Table 2 provides a sampling of tools available to perform this type of technique.

These tools let the observer see the interplay *between* components. This interplay is not limited to software component interfaces, but includes hardware interfaces as well. In fact, snooping can occur at any point where data and control extend past a single component's boundary. This can include:

- inter-component communication across process and processor boundaries;
- parent/child communication;
- remote procedure calls;
- client/server communication; and
- dynamic data exchange.

Similarly, such snooping is not limited to network traffic (e.g., Ethernet, FDDI), but can be used on other physical transport layers such as RS-232 or SCSI. Snooping techniques can thus improve understanding of the external behaviour and interfaces of an off-the-shelf component and its interplay with other components within the system.

4.4. Extra-component visibility

Techniques in this category provide insight into the overall workings of the integrated components from the system-level or subsystem-level perspective. The general technique involves finding

objective measurements about the functioning of the system. These include static or dynamic data about performance, system throughput, resource utilization and response time. These data are found in audit logs, error files and similar sources.

The principal approach is to capture and analyse system-level behaviour. The data that are obtained can be used for early detection, causal analysis and intervention. This information is more valuable when data on the system (and appropriate subsystems) are collected over time, thus establishing a historical portrait for the system. As components are upgraded in the system, fluctuations from historical norms (particularly negative fluctuations) can be an early indicator of failure. For example, suppose that a network server for a corporate LAN (local area network) is being upgraded to a computer with parallel processors. On completion of the upgrade, data collected after installation show a downward trend in server response time (i.e., compared with the pre-upgrade historical data). Since the realized performance impact was less than expected, some further inspection of the system's behaviour is called for. It is also true that the same kind of data can be an indicator that the upgrade has improved the system, and the degree to which the change was positive. Essentially, these techniques improve the ability to quantify objectively and assess changes in the system, measure the effects of those changes, and optimize time and resources needed for debugging.

5. THREE EXAMPLES OF USES OF THE SCIENTIFIC METHOD

5.1. An example of intra-component visibility: debugging a COTS-based information system

Our first example of how integrators used the scientific method in debugging COTS-based systems is an information system that provided dynamic access through the world-wide web (WWW) to data and information contained in a commercial database. The system was comprised of a commercial WWW server and a commercial relational database management system (RDBMS) operating on a UNIX platform. For communication between the web server and the RDBMS we depended on a product from the database vendor called a 'server agent'. This COTS product made use of both a proprietary interface to the RDBMS and a proprietary interface to the web server. In addition, this 'server agent' could support the publicly defined common gateway interface (CGI). However, we chose to use the proprietary interface since it provided faster execution time and was more efficient in system resources. Figure 2 diagrams the overall system. This system has been described in greater detail as a case study by Hissam (1998, Section 3).

Late in the development cycle it was noticed that system performance was extremely poor, for no apparent cause. To correct this poor performance, we made four iterations through the 'observation–hypothesis–prediction–experiment' cycle. In the first iteration, the execution of the RDBMS was confirmed as the performance bottleneck. This was accomplished through creation of a simple harness that launched skeleton browser processes and compared the observed latency time with the users' recorded reports of poor performance. Now our attention turned to why.

We based our second hypothesis on the assumption that in a multi-user system like this, child processes would be spawned to accommodate new queries. Our hypothesis was that these supposedly spawned processes might not actually exist. Using the test harness from the first

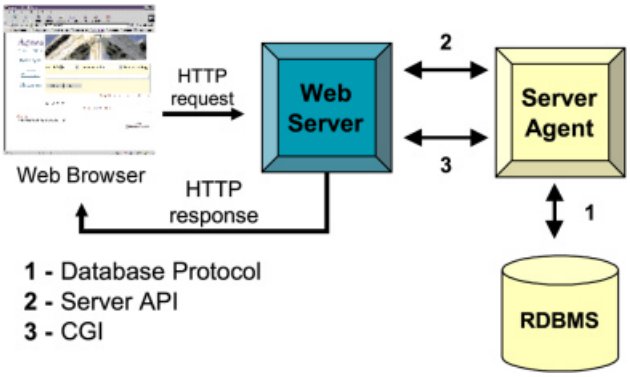


Figure 2. Web server/database diagram

```
$ ps -ef
UID    PID   PPID   C   STIME  TTY      TIME COMD
              :
nobody 11379 [illegible]  3   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11380 [illegible] 47   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11381 11379 20   Jun 24 ?        0:20 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11382 11379 20   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
nobody 11383 11379 32   Jun 24 ?        0:00 ./httpd -d /export/home2/home/httpd-jc/config
              :
```

Figure 3. Output from `ps`

iteration, we probed with the UNIX `ps` command to observe the state of all processes. If our hypothesis was true, then spawned processes would not appear. However, this prediction was not borne out, since several other processes were observed. The output from the `ps` tool is shown in Figure 3.

Two additional observations resulted from this iteration. First, we noted that the command line argument to the WWW server was a directory containing the configuration settings for the WWW server. One of these was a tunable parameter for the number of child processes (in this case, set to four, corresponding to PIDs #11381 to #11383 shown in Figure 3). The more interesting observation was that one of the children (PID #11381) had accumulated 20 seconds of CPU time while the others had none.

Our third hypothesis was a direct result of this: perhaps some of the child processes were not handling requests. If so, then some of these should be idle while others were working (and based on the previous iteration, then all but one should be idle.) We therefore used the UNIX `truss` command to gain insight into what each spawned process was actually doing. We learned that these child processes were not in fact idle. They were all making a specific system call (`fcntl`), returning an error status of `EAGAIN` and trying again. This meant that all of these child processes were attempting to set a lock (i.e., `F_SETLK`) on some resource that was not available. (`truss` output is illustrated in Figures 4 and 5).

When the healthy child terminated, the parent spawned another child process, and a race condition would then exist between the new child and the three existing children; one would set the lock and

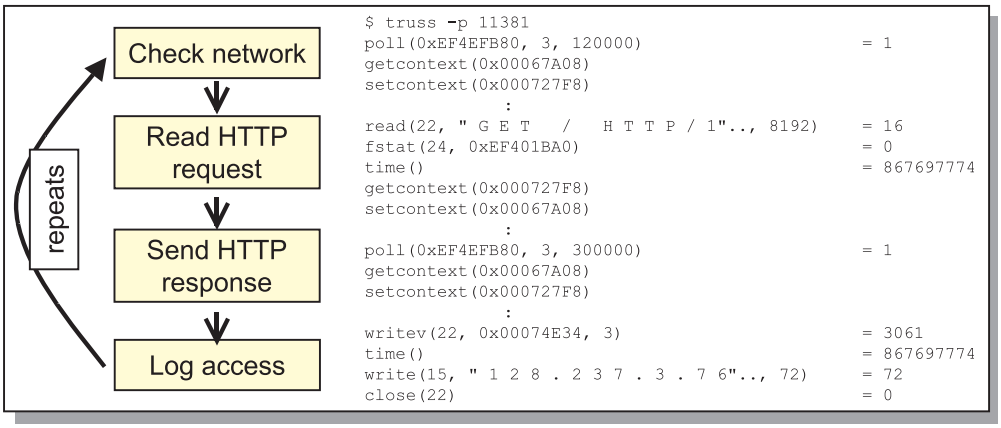


Figure 4. WWW server child working state

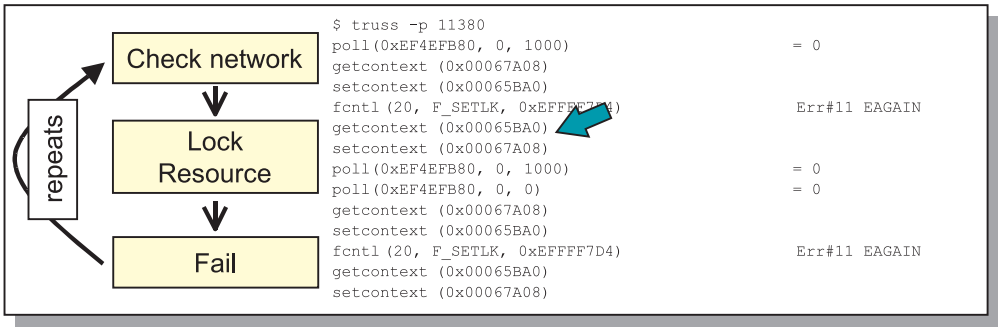


Figure 5. WWW server child failing state

the remaining three would again fall into the status shown below in Figure 5.

There was one other critical result from this iteration: we also noted that the WWW server was making use of the system calls `getcontext` and `setcontext` (see Figure 5). These calls are used in user-level threaded applications, which implied that the WWW server used user-level context switching between multiple threads of control. This led to the fourth—and successful—hypothesis.

As noted earlier in the description of the overall system, we had chosen to use the proprietary API defined for the COTS product. We now investigated the implications of this decision. First, using one protocol over another has implications for the execution environment in which the WWW server and the server agent will run. In the proprietary interface, the server agent shares the address space with the server; by contrast, the CGI protocol assumes a separate address space. Second, the proprietary interface is tightly coupled, and assumes that once loaded, communication will be like many other tightly coupled application/library interfaces. For CGI applications, communication is loosely coupled, relying heavily on operating system services for interprocess communication.

We therefore surmised that the server agent was not thread safe and was causing the WWW

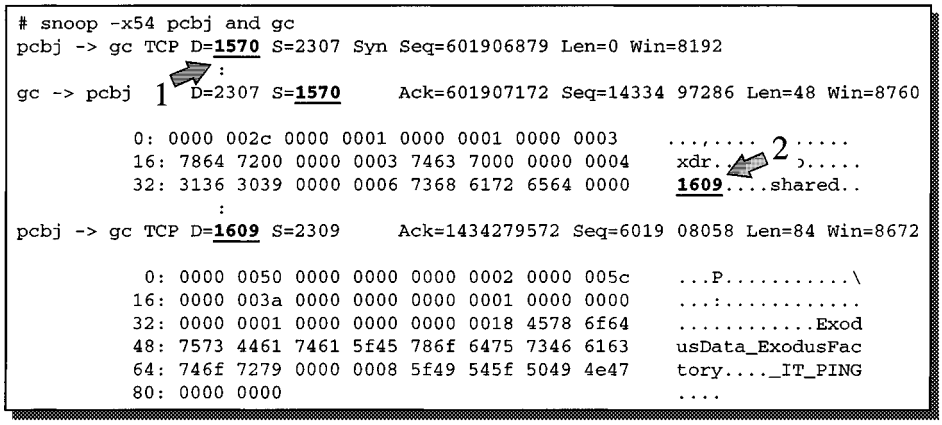


Figure 6. Snooping healthy legacy application

server to block; our hypothesis was that replacing the proprietary API with the CGI interface would solve the blocking problem. We did this through reconfiguring and reinstalling the server/agent. The hypothesis turned out to be correct, and the system now functioned as expected, as illustrated in Figure 2.

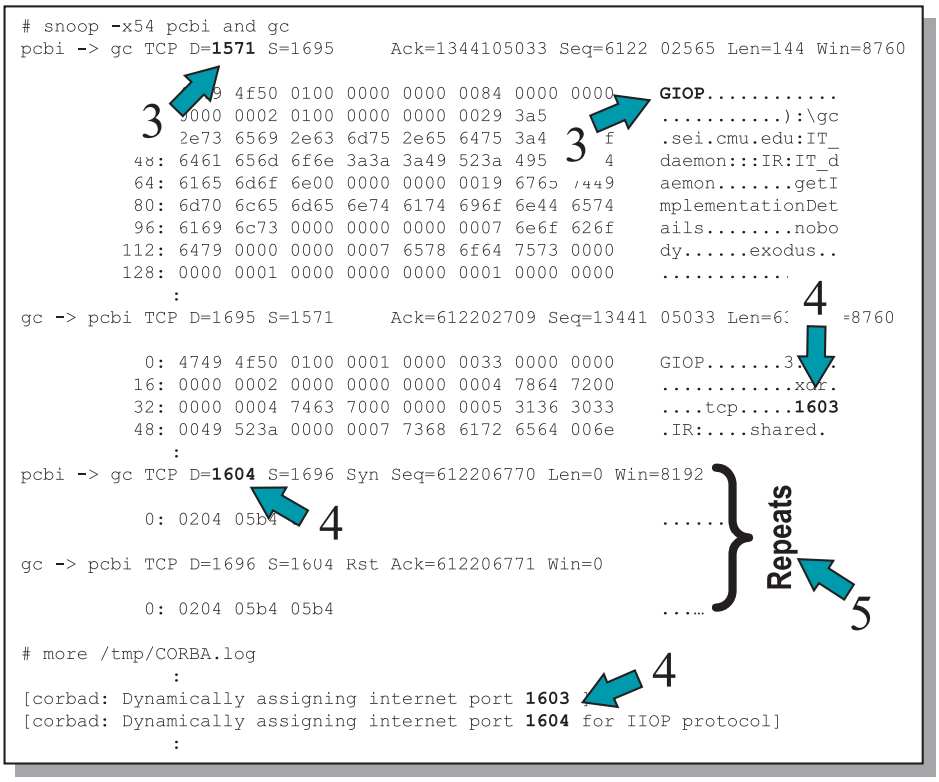
5.2. An example of inter-component visibility: debugging a Java applet and CORBA server application

The system under investigation was a simple client/server application where a Java applet[†] communicated with a CORBA[‡]-based server. The applet was to be loaded by a Java-enabled WWW browser, display a graphical-user interface (GUI), establish an IIOP[§] connection to the server and invoke operations provided by that server. This system consisted of a legacy server for which a new front-end client was being developed. It was discovered during early development of this Java application that communication with the server was failing while the legacy client was continuing to operate properly.

An initial hypothesis was formed that the Java applet was not establishing a connection to the legacy server via IIOP or that the IIOP protocol between the client and server was failing. We predicted that by observing the network traffic between the legacy client and the server we would see the client and server using the proprietary communications protocol while the new Java client's attempts at communication via IIOP would either go unanswered or appear to fail.

To test this prediction, we used a UNIX network analysis tool called `snoop`. In the first stage of the experiment we isolated the network traffic that was present during a healthy session between the legacy client and server. Figure 6 shows some of what was observed during that experiment.

[†]Java—more information on Java™ can be found at `http://www.javasoft.com`. Applet is a term used to describe a thin client application that is loaded on demand by a Java-enabled browser such as the Microsoft™ Internet Explorer or the Netscape™ Navigator or Communicator browsers.
[‡]CORBA—Common object request broker architecture: more information can be found at `http://www.omg.org`.
[§]IIOP—CORBA/internet inter-ORB operability protocol: more information can be found at `http://www.omg.org/corba/corbiiop.htm`.




```
# snoop -x54 pcbi and gc
pcbi -> gc TCP D=1571 S=1695     Ack=1344105033 Seq=6122 02565 Len=144 Win=8760

           0: 4749 4f50 0100 0000 0000 0084 0000 0000    GIOP............
          16: 0000 0002 0100 0000 0000 0029 3a5         ...........):\gc
          32: 2e73 6569 2e63 6d75 2e65 6475 3a4  f      .sei.cmu.edu:IT_
          48: 6461 656d 6f6e 3a3a 3a49 523a 495  4      daemon:::IR:IT_d
          64: 6165 6d6f 6e00 0000 0000 0019 676c /449    aemon.......getI
          80: 6d70 6c65 6d65 6e74 6174 696f 6e44 6574    mplementationDet
          96: 6169 6c73 0000 0000 0000 0007 6e6f 626f    ails........nobo
         112: 6479 0000 0000 0007 6578 6f64 7573 0000    dy......exodus..
         128: 0000 0001 0000 0000 0000 0001 0000 0000    ............
                :
gc -> pcbi TCP D=1695 S=1571     Ack=612202709 Seq=13441 05033 Len=6?   =8760

           0: 4749 4f50 0100 0001 0000 0033 0000 0000    GIOP.......3....
          16: 0000 0002 0000 0000 0000 0004 7864 7200    ............xdr.
          32: 0000 0004 7463 7000 0000 0005 3136 3033    ....tcp.....1603
          48: 0049 523a 0000 0007 7368 6172 6564 006e    .IR:....shared.
                :
pcbi -> gc TCP D=1604 S=1696 Syn Seq=612206770 Len=0 Win=8192

           0: 0204 05b4                                  ....

gc -> pcbi TCP D=1696 S=1604 Rst Ack=612206771 Win=0

           0: 0204 05b4 05b4                             ....

# more /tmp/CORBA.log
                :
[corbad: Dynamically assigning internet port 1603 f
[corbad: Dynamically assigning internet port 1604 for IIOP protocol]
                :
```

Figure 7. Snooping failing application

There are two significant points observed in this figure:

1. the initial communication with the CORBA services occurs on TCP port *1570*, which for this system is the well-advertised port number for this vendor's proprietary communications protocol; and,
2. the data returned from the server host show, as can be observed from item 2 in Figure 6, that TCP port number *1609* is used in subsequent communication between the legacy client and server.

With knowledge of the healthy system, the second stage of this experiment was to observe the network traffic for the failing portion of the system. The same tools were used and some of the observations recorded are depicted in Figure 7.

There are three significant points observed in this figure:

1. The initial communication with the CORBA services occurs on TCP port *1571*, which for this system is the well-advertised port number for the CORBA/IIOP communications protocol. Use of IIOP is confirmed through the `'GIOP'` marker found in the data stream (which is consistent with the CORBA/IIOP specification).

2. As can be observed through item 4 in Figure 7, data returned from the server host show that TCP port number *1603* is returned but *1604* is actually used for subsequent communications. According to the log file generated by the CORBA daemon, *1604* is the port number specifically assigned to the legacy server for CORBA/IIOP connections.
3. Item 5 in Figure 7 shows that subsequent attempts to communicate on TCP port *1604* repeat and then fail once the time-out threshold is reached. The Java applet then terminates.

It was clear from the experiments and the resultant observations that the legacy server was refusing to operate on the assigned CORBA/IIOP port as expected by the CORBA specification. The problem lay not with the newly developed Java applet or Java class libraries but with the legacy server. This information was provided to the maintainers of the legacy server who then made proper configuration changes to permit the legacy server to inter-operate with the Java applet via the CORBA/IIOP specification.

5.3. An example of extra-component visibility: debugging a distributed relational database system

The system in question was a typical distributed three-tiered application that used user-level components, middle-layer servers containing application-specific logic, business rules and COTS wrappers, and back-end relational database servers (RDBMS). The system served developers, testers and users from the operational area as well. Based on user input, the middle-layer servers would make queries to the RDBMS, aggregate the resultant data sets and summarize those results for the user. After deployment, an intermittent error message appeared that could not be consistently reproduced: if the user resubmitted a failing request, it might or might not occur again.

Our first hypothesis was that the code in the middle tier was responsible for the error. Since these modules had been developed in-house, they were naturally believed to be a likely source of error. Exhaustive searches of the available source code and binary searches of middleware components failed to locate the error message that was intermittently generated. This silenced the first hypothesis.

Our next hypothesis was that the error message was coming from the RDBMS and was passing through the middle tier. Although a viable hypothesis, the team responsible for the RDBMS was reluctant to investigate or entertain this notion. Without the co-operation of the RDBMS team, we decided to treat the back-end relation database layer as a commercial product and gather sufficient evidence to convince the RBDMS team that the fault lay there.

Since previous attempts to reproduce the errors consistently had failed, we needed to take a system-wide view to locate the source of the error. We therefore began a detailed analysis of the available system logs and error reports for our initial observation. Log files from five different middle-level servers were analysed. Data from each of these servers were plotted against the date on which the error occurred; the result is shown in Figure 8.

Some significant patterns of behaviour emerged:

1. `Host 3, Host 5` and `Host 2` (we will call these server group 1, or SG1) showed a significant correlation of the error on four dates;

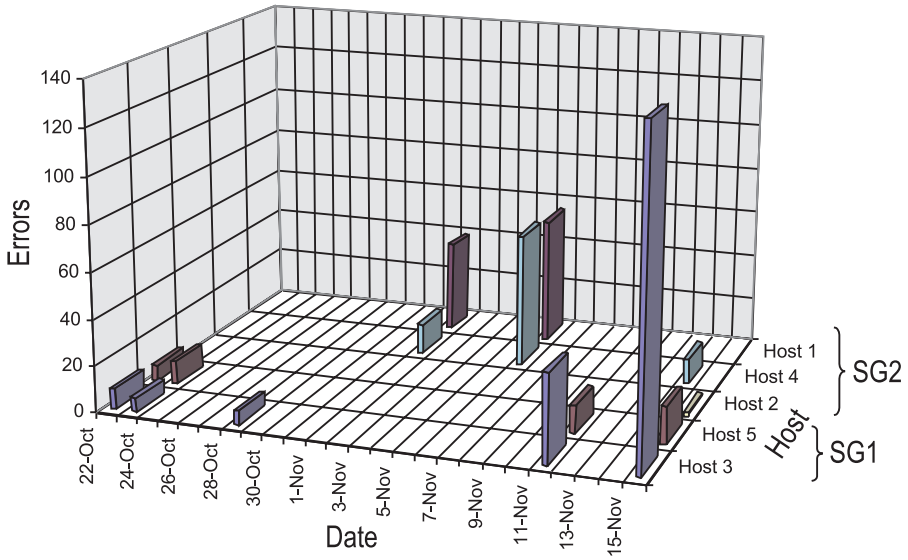


Figure 8. Distribution of errors by data and by server

2. `Host 4 and Host 1` (server group 2, or SG2) showed a comparable correlation for the error on three different dates; and
3. errors were thus clustered to either server group 1 or server group 2.

Next we correlated this pattern of errors against the known database configuration. This configuration is shown in Figure 9. From this our working hypothesis was that the relational databases were responsible for the intermittent errors. What was necessary at this point was to provide sufficient evidence to support this hypothesis. What the data from the analysis show is that the servers of SG1 and SG2 were failing in the same manner, but at different times. Since each group was isolated from each other, the stimulus that provoked the failure was chronologically unique to each server group (i.e., SG1 was affected by stimulus1, SG2 was affected by stimulus2). In Figure 9, it can be seen that each of the server groups interact with users (at the top tier) and RDBMS (at the lower tier). Since the error originated after the query was submitted to the RDBMS, the evidence was strong enough to warrant an in-depth investigation into the lower tier (the RDBMS).

As can be seen, there is a distinct parallel between the pattern of behaviour we noted and the high-level system configuration. Each portion of the system used an independent data source for the lower tier; in one case it was an operational RDBMS, in the other a development RDBMS. This now strongly indicated that the resource constraint that was causing the error somehow lay with these databases.

Based on this evidence, the RDBMS team examined the lower tier RDBMS, searching for resource constraints and erroneous conditions. They eventually determined that temporary database table space was being exhausted as the databases responded to queries after long periods of time. These failures were dependent on high load factors (thus explaining the inconsistency), and the resultant failures cascaded through the entire system.

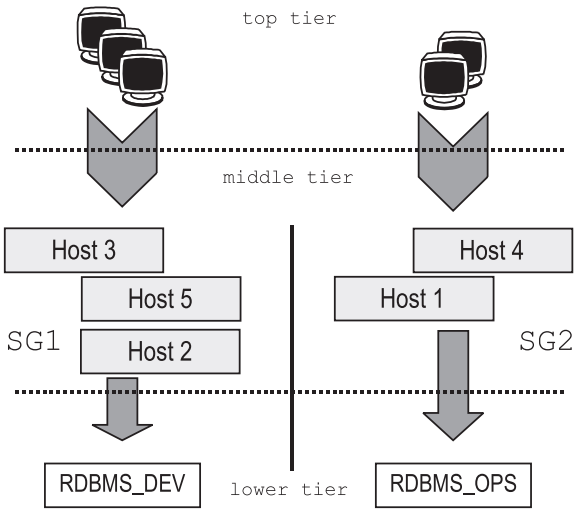


Figure 9. High-level system configuration

6. LESSONS LEARNED

These experiences provided a number of valuable lessons, of which the following are most important.

1. For detecting faults in a COTS-based system, the importance of a methodical process cannot be overestimated.

Without a structured and repeatable process (whether the one we followed or some other), it is doubtful that we would have achieved the success that we did. There are simply too many unknowns when the source code is lacking, and the number of potential interactions between even a small number of COTS products makes the problem space very large. We found that the only way to search that problem space was through application of the scientific method.

2. In developing hypotheses about faults, removing as many variables as possible was useful.

The lesson we learned was simplicity. While it can be tempting to try to determine many things about how COTS products work, and formulate large-scale hypotheses about them, we found it more useful to develop hypotheses that were essentially simple, and that aimed at determining one specific piece of data about the product.

3. A taxonomy is extremely useful for forming hypotheses.

One of the most useful aids in this work was the separation of the problem space (i.e., into inter-, intra- and extra-component visibility). However, a richer taxonomy is needed—different factors indicating configuration or installation errors as opposed to internal coding errors; or heuristics to separate incompatibilities between component versions versus specification mismatch. Either of these would be useful in performing this task.

4. Deep expertise—both about the underlying system and the available debugging tools—is necessary.

Debugging systems like those described above is probably not possible for maintainers who are unfamiliar with low-level system internals. We also found that broad experience could be helpful in many unexpected ways, as when we serendipitously noticed how some low-level calls indicated that user-level threads were being used by one of the components (cf. Section 5.1).

5. Gaining visibility at each level of the system—intra-component, inter-component and extra-component—has specific value and benefits.

We found the following benefits to be true:

Intra-component tools and visibility:

- help the integrator to create detailed problem reports for the component's developer, thus providing the evidence needed to convince the developer to address the problem quickly;
- place the integrator on a more level playing field in terms of knowledge and evidence with the developer; thus improving communication;
- increase the integrator's understanding of the underlying assumptions and design characteristics used by the component developer; and
- enhance ability of the integrator to quickly evaluate new releases or component repairs.

Inter-component tools and visibility:

- isolate the point of failure to a particular component pairing;
- identify deviations from standards; and
- provide understanding about component external interfaces (such understanding also provides a potential entry point for adapting a component).

Extra-component tools and visibility:

- isolate system/subsystem (as well as individual component) failure points;
- calculate the impact of new releases on performance (i.e., expected versus realized); and
- provide performance tuning hints.

7. CONCLUSION

The ever-increasing prevalence of COTS in today's systems is bringing new challenges to systems development and maintenance as we know it. By their very nature, COTS components are difficult to evaluate carefully, are opaque to the integrator and are especially not susceptible to traditional source-code-based debugging. While newly evolving evaluation techniques and improving vendor support are hopeful signs, ultimately the onus is on the system integrator to diagnose and solve problems when things go wrong.

As we have shown in this paper, the scientific method is applicable to software engineering, and perhaps is especially applicable in those systems involving COTS components. As we have also demonstrated, observation is the key to this method, and obtaining useful observation into COTS components is often more challenging than traditional debugging.

Sophisticated tools exist to aid the integrator in building the observational evidence needed for system repair and in establishing a better understanding of the COTS component and its underlying assumptions. However, a structured method for using these tools is needed. We have described one such method in this paper. In addition to this method and tools, the integrator will need technical prowess, diagnostic and deductive insight, and a willingness to try many approaches to successfully maintain complex systems using a mixture of commercial and custom components.

References

Beizer, B. (1995) *Black-box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, Inc., New York NY, 294 pp.

Carney, D. and Wallnau, K. (1998) 'A basis for evaluation of commercial software', *Information and Software Technology*, **40**(14), 851–860.

Chan, T. W. and Lakhotia, A. (1998) 'Debugging program failure exhibited by voluminous data', *Journal of Software Maintenance*, **10**(2), 111–150.

Fox, G., Lantner, K. and Marcom, S. (1997) 'A software development process for COTS-based information system infrastructure', in *Proceedings Fifth International Symposium on Assessment of Software Tools*, IEEE Computer Society Press, Los Alamitos CA, pp. 133–142.

Hissam, S. A. (1998) 'Experience report: correcting system failure in a COTS information system', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 170–176.

Hybertson, D. W., Ta, A. D. and Thomas, W. M. (1997) 'Maintenance of COTS-intensive software systems', *Journal of Software Maintenance*, **9**(4), 203–216.

Myers, G. J. (1997) *The Art of Software Testing*, John Wiley & Sons, Inc., New York NY, 177 pp.

Toulmin, S. E. (1987) 'Philosophy of science', in *Encyclopedia Britannica, Vol. 25*, Encyclopedia Britannica, Inc., Chicago IL, pp. 661–678.

Voas, J. (1998) 'The challenges of using COTS software in component-based development', *IEEE Computer*, **31**(6), 44–45.

Authors' biographies:

Scott A. Hissam is a Member of the Technical Staff in the COTS-Based Systems initiative at the Software Engineering Institute (SEI) at Carnegie Mellon University. He is currently working with US Federal Department of Defense (DoD) programs in the use of distributed object technology and COTS products. Before joining the SEI, Scott worked for Lockheed Martin as Senior Engineer for the Global Transportation Network's WWW interface and as Program Manager of the National Software Data and Information Repository. Prior to that, he was employed by Bell Atlantic and earlier as a Computer Analyst for the US Department of Defense. Scott has a B.S. in Computer Science from West Virginia University. His email address is: shissam@sei.cmu.edu

David Carney is a Senior Member of the Technical Staff in the Dynamic Systems program at the Software Engineering Institute (SEI) at Carnegie Mellon University. He is currently working with the US Federal DARPA program on Evolutionary Design of Complex Software, and has participated in projects with DoD, FAA, and other Federal government agencies. Before joining the SEI, David was on the staff of the Institute for Defense Analyses in Alexandria VA, where he worked with the STARS Program and with the NATO Special Working Group on APSE. Prior to that, he was employed at Intermetrics, Inc. and worked on the Ada Integrated Environment project. David holds an M.S. in Computer Science from Boston University. His email address is: djc@sei.cmu.edu

